A Software Bus for Thread Objects

John Callahan[1]
Dehuai Li
Department of Statistics and Computer Science
Concurrent Engineering Research Center
West Virginia University

Abstract

We have implemented a software bus [1] for lightweight threads in an object-oriented programming environment that allows for rapid recofniguration and reuse of thread objects in discrete-event simulation experiments. While previous research in object-oriented, parallel programming environments have focused on direct communication between threads, our lightweight software bus, called the MINIBUS, provides a means to isolate threads from their contexts of execution by restricting communications between threads to message-passing via their local ports only. The software bus maintains a topology of connections between these ports. It routes, queues, and delivers messages according to this topology. This approach allows for rapid reconfiguration and reuse of thread objects in other systems without making changes to the specifications or source code. We present a layered approach that provides the needed transparency to developers, discuss example cases of using the MINIBUS, and the value of bus architectures in building and conducting simulations of discrete-event systems.

Catagory: research
Topic Areas: tools and environments, components and frameworks, concurrent systems

1 Introduction

Expressive programming languages and run-time mechanisms are needed to implement message-passing architectures between processes in parallel and distributed computing environments. Most programming languages provide abstractions like procedure call and method invocation to implement synchronous communications between components in sequential programs. In parallel and distributed computing environments, however, indirect and asynchronous mechanisms are also needed to provide communications between multiple, independently executing processes. Such mechanisms permit rapid reconfigurations of systems since each component is highly independent of other processes.

---

[1]Please contact John R. Callahan, Assistant Professor, Department of Statistics and Computer Science, West Virginia University, Morgantown, WV 26506-6330, Phone: 304-293-3607, Fax: 304-293-7541, callahan@cs.wvu.edu. This work is supported by NASA Grant NAG 5-2129, NLM Grant N01-LM-3-3525, and ARPA Grant MDA 972-91-J-1022.

A *software bus* [1] is an abstraction for distributed and parallel programming environments that provides for indirect and asynchronous communications between processes. A software bus routes, queues, and delivers messages between client processes and provides a standard set of communication primitives to all client processes. Each client process is viewed as a self-contained module that owns a unique, local set of "bus ports" on which it can send and receive messages. All communication is performed on local ports only. A bus application consists of a set of client processes and a "bus" that maintains a map of connections between client ports. A message sent on port X of process A is delivered by the bus to port Y of process B if the connection A.X $\rightarrow$ B.Y is part of the bus topology. Software bus systems share much in common with event-based approaches, but the software bus abstraction subsumes event-based models by allowing more structured forms of interactions between processes. Events can be simulated in the bus topology by routing all send ports to all receive ports.

We have implemented a version of the POLYLITH software bus for lightweight threads based on the AWESIME [2] class library. The AWESIME class library is implemented in C++ and provides a parallel programming environment for lightweight threads in a single address space. Threads in AWESIME are created as instances of derived subclasses of the abstract `Thread` class. Physical processors are simulated by a `CPU` class that can contain thread instances. Thread instances are placed into CPU instances and the CPU is started. At this point, each CPU instance schedules and executes its threads until a termination condition. There are several subclasses of `CPU` (e.g., MULTIFIFOMUX) and each subclass schedules execution of its threads according to different scheduling algorithms.

Our implementation of a lightweight software bus, called the MINIBUS, provides a mechanism for indirect and synchronous communication between threads. Normally, AWESIME threads communicate by directly invoking methods of other `Thread` instances, through shared variables or by direct message passing primitives. This means that threads are highly coupled to their context of execution with other threads. The systems in which they are embedded are not easily reconfigured and the threads themselves not reusable in other programs. In the MINIBUS, threads are viewed as self-contained modules that invoke their own virtual methods to communicate through the software bus. These methods are local to instances of the Thread subclass and implement local ports on which a thread may send and receive messages. A separate lightweight process acts as the bus to route and queue messages between threads. This approach allows for rapid reconfigurations and reuse of threads in other applications because indirect communication eliminates any coupling between threads and shared variables.

Previous thread-based, parallel programming environments based on object-oriented languages [3,4] have focused primarily on direct communications between threads in a fashion similar to CSP [5] or an events [6]. While this is useful for some problems, the direct communications between threads is usually accomplished by addressing recipients explicitly in the source code of the sender. This approach,

however, couples component designs to each other. If the context of execution changes, e.g., a different receiver thread is used, the sender source code must be updated. In some cases, message format differences, the use of broadcast or multicast of messages, and changes to the numbers of threads means that a number of changes are needed to many threads as a result of the high degree of coupling. Many components must be reimplemented because of changes to other components and changes to the structure of the system in which a component is embedded.

This paper describes the application programmers interface (API) of the MINIBUS and shows how parallel programs are constructed so that reconfigurations have little or no impact on the specification or implementation of thread components. We illustrate the effectiveness and capabilities of the MINIBUS in building simulations by reusing components in several applications and rapidly reconfiguring existing applications with no effect on the source code of the thread components.

2 Overview

When two threads communicate directly with each other, each process must know exactly what mechanism the other thread uses to enable communication. The design of each thread is coupled to others in a configuration. A software bus provides a standard set of communication channels that allows each thread to be fully encapsulated from its context of execution, thus reducing the complexity of the design problem, and increasing code independence and reuse. In the software bus model, a thread sends messages to the bus via local ports rather than directly to the intended receivers. The bus then routes the message to its destination(s). There is no direct interaction between the sending and the receiving threads.

Figure 1 illustrates communications between processes in a software bus application. Although the processes execute independently, they all interact with the bus only through their local ports. Processes A, B, and C may send and receive message only on their local ports. The bus routes messages between ports depending on a given topology of connections. This topology may be changed, connections may be made from a single port to multiple ports, and a single receive port can accept messages from multiple incoming connections. The ability to "rewire" an application is a major advantage of the software bus model. Processes can be added and removed from configurations, often during system execution, without changing the internal code of individual processes [7].

3 The MINIBUS Software Bus

In the AWESIME parallel programming environment, a thread is the basic unit of parallelism. A thread is an instance of a derived subclass of the C++ abstract class `Thread`. Developers create subclasses of

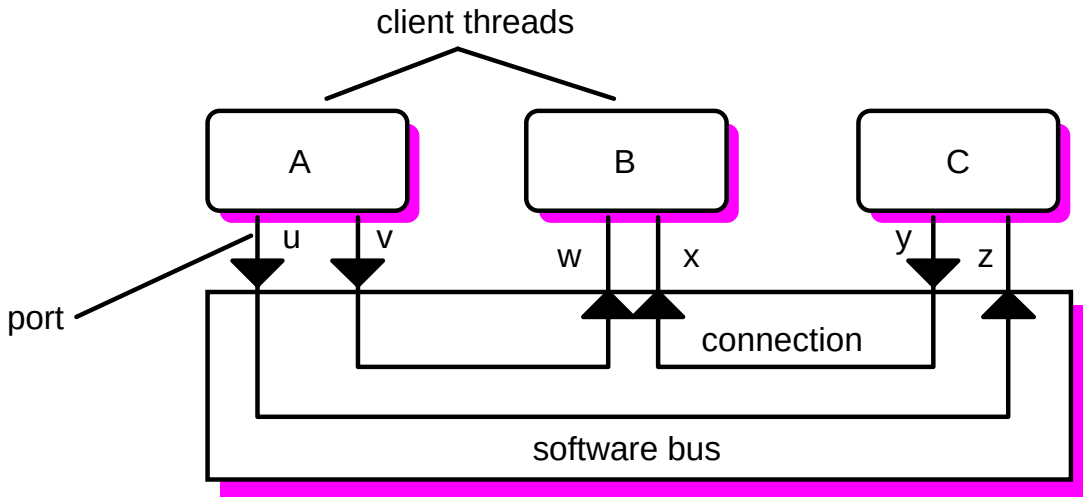

Figure 1: Communication between processes through a software bus

`Thread` for each type of application thread. A typical application in AWESIME is composed of a set of instances of classes derived from the base class `Thread`. Each type of derived class may be used to instantiate any number of threads in the application

AWESIME does not provide a general mechanism for communication between threads except through direct method invocation, shared variables, and direct message passing. We have implemented a software bus that provides an indirect, asynchronous communication mechanism between threads. In the bus model, an additional level of subclassing is needed to implement our approach. Figure 2 illustrates the levels in a class hierarchy needed to construct a layered communications model based on a software bus. The top level `Thread` class is provided in the AWESIME package, application subclasses of `Thread` in the middle layer define the specialized functionality of each thread type (i.e., the behavior of a thread is implemented in this layer), and the interfacing classes in the bottom layer implement low-level communications with the bus. Local ports to the bus are specified as pure virtual methods in application class specifications that are implemented by interfacing subclasses. When a thread sends a message, its invokes one of its own virtual methods whose implementation is found in its interfacing subclass. This approach allows developers a high degree of flexibility in substituting different communication technologies at the interfacing level while leaving the application level thread code unchanged. Indeed, during development of the MINIBUS we changed the bus API several times with little or no effect to thread code in the application layer. We discuss the details of our layered approach in the next sections.

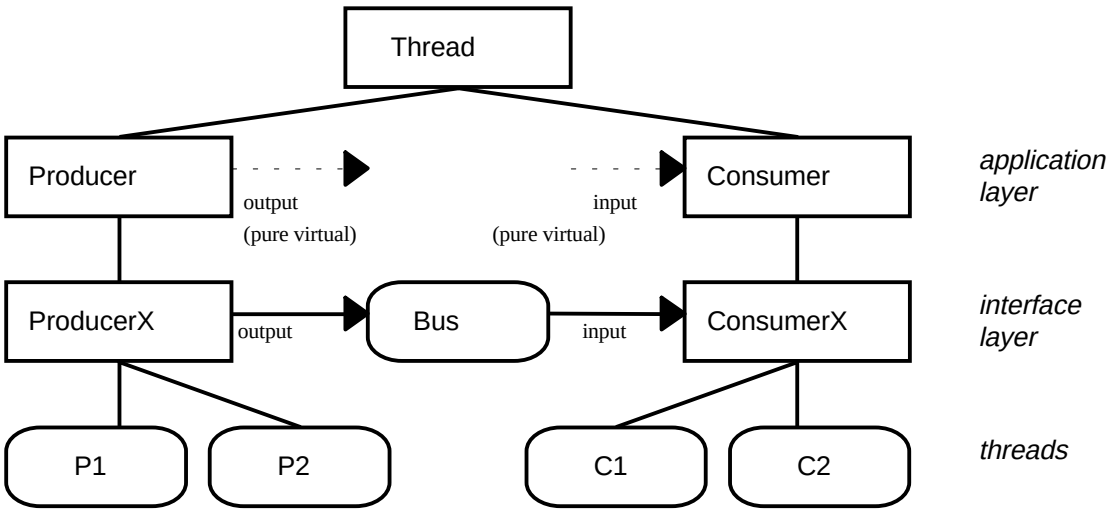

Figure 2: Layered approach to communications between threads via a software bus

3.1 The Thread Class

The abstract class `Thread` is the base class for all threads in an application. Most of the details of the `Thread` class are uninteresting to the casual user. All application threads must be instances of a subclass of class `Thread.` Threads should only be created using the C++ `new` operator because the AWESIME run-time environment manages their stacks on a dynamic heap. In addition, each subclass of class `Thread` should provide implementations to support three functions: a constructor, a destructor, and a main entry point function. First, each subclass of `Thread` must support its own constructor that calls the `Thread` constructor with initialization arguments. Second, each subclass of `Thread` must implement a destructor that cleans up any data in the class instance. Finally, each subclass of `Thread` must implement a function called `main` defined as a virtual function in `Thread`. The `main` function is the entry point of all threads created from the application subclass. When the thread is executed for the first time, the `main` function is called by the `CPU` instance that contains the thread.

3.2 The Application Classes

All application classes are derived from the class `Thread` and form a level of abstract classes in the class hierarchy called the application layer. Classes in this layer specify the functional behavior of threads in the application. Application classes must provide implementations for a constructor, destructor, and main functions. An application class specification must also specify all of its local send and receive ports as pure virtual functions.

In Figures 3 and 4, we show the specification and implementation for the `Producer` class. The `Producer` constructor will intialize the parent `Thread` class with a thread name, a maximum stack size of 20000 words, a hardware memory flag (HCCM), and set the priority of any `Producer` threads equal to the process identifier assigned to them. It also initializes the private member variable `maxsend` with the maximum number of messages to produce. The implementation of the `Producer::main` method is the main loop of any `Producer` thread instance. It sends `maxsend` number of messages and then terminates. The messages are sent by calling the pure virtual function `output` that has no associated implementation at this level or above in the class hierachy. Such functions are implemented in subclasses of `Producer` in the interfacing layer.

3.3 The Interfacing Layer

The classes in the interfacing layer are derived from classes in the application layer. The interfacing classes are the concrete classes from which thread instances are created. Each concrete class in the interfacing layer implements the virtual functions of its parent class that represent the local ports of a thread type. Figures 5 and 6 show the specification and implementation of the interfacing class `ProducerX` derived from the Producer application class. The `ProducerX` constructor invokes the `bus->init` method that initializes the thread with the bus thread. The output implementation sends a message to the bus process via the method

```
bus->write("output","I",NULL,NULL,i)
```

where the first argument "output" is the name of the local port and the second argument is a description of the types of data object passed in the message (i.e., "I" stands for a single integer). The arguments given NULL in this case can be used to transfer port capabilities between threads. Finally, the value of the variable `i` is sent in the body of the message.

A `Consumer` subclass of `Thread` and a `ConsumerX` subclass can be constructed in a similar fashion. In the case of a consumer thread, a call to a virtual function `input` within the main loop of the `Consumer::main` body will invoke the implementation of `input` in a `ConsumerX` subclass. As in the case of the `ProducerX::output` function, the `ConsumerX::input` would be implemented by a complementary call to receive a message via the method

```
bus->read("input","i",NULL,NULL,&r0)
```

that reads the next message on the local input port of the `ConsumerX` thread or blocks if the message queue is empty. The second parameter of the `bus->read` function, as in the `bus->write` function,

```
class Producer : public Thread {
     int sent;
     int max;
public:
     Producer(char *name,int xpid,int maxsend);
     ~Producer();
     void main();
     virtual void output(int);
};
```

Figure 3: Producer abstract class specification

```
Producer::Producer(char *name,int xpid,int maxsend)
     : (name,20000,HCCM,xpid)
{
     max = maxsend;
}

void Producer::main()
{
     for(int i=0;i < maxsend;i++) {
          CERR_ALWAYS_PRE;
          cout << "Produce " << i << endl;
          CERR_POST;
          output(i);
     }
}
```

Figure 4: Implementation of Producer methods

```
class ProducerX : public Producer {
     Bus* bus;
public:
     ProducerX(Bus*,char*,int,int);
     void output(int);
}
```

Figure 5: ProducerX concrete class specification

```
ProducerX::ProducerX(Bus* b,char* name,int xpid,int max)
     : (name,xpid,max)
{
     bus = b;
     bus->init(this);
}

void ProducerX::output(int a1) {
     bus->write(this,"output","I",NULL,NULL,a1);
}
```

Figure 6: Implementation of output method for ProducerX class

represents the format of data expected to read. The data contained in the received message (an integer) is read into variable `r0` and returned as the result of the call to the virtual function `input`.

Using this indirect approach to communication between lightweight threads, the implementations of all threads are completely encapsulated from their context of execution. The thread application classes can be reconfigured and reused in new applications with no changes needed to their sources. All messages are routed, queued, and delivered by the bus thread. Section 4 describes the implementation of the bus thread.

3.4 The main function

Figure 7 shows the main entry point for the producer-consumer application. The C++ run-time environment first executes the `main` function when the program starts. The program declares pointers to two threads for the producer and consumer and then declares an instance of `MultiFifoMux` class (an AWESIME subclass of the CPU abstract class) that simulates multiple CPUs. It then creates a bus thread and adds the bus thread to the CPU instance. Next, the program creates thread instances of the `ProducerX` and `ConsumerX` classes and adds each thread instance to the CPU. The `bus->connect` method then adds a connection to the bus topology. This connection states that any message sent by the producer thread on its output port should be routed to the input port of the consumer thread. Finally, the CPU instance is started via a call to its `fireItUp` method. The CPU instance then takes control of program execution at this point until both of its threads have terminated.

4 The Awesime2Bus Bus

The bus thread is created by the main program and executes at a higher priority than other threads in the application. It functions much the same as a post office with one major differemce: it collects messages and places them into the mailboxes (i.e., queues) of intended receivers, but routes the messages according to its toplogy not an address associated with each message. This topology may change dynamically during execution of the application. The bus provides methods for creating connections between thread ports as well as writing and reading via local ports. The next sections describe the details of low-level bus operations.

```
main()
{
     Thread* p,c;
     MultiFifoMux cpu;

     Bus bus = new Bus();
     cpu.add(bus);

     cpu.add(p = new ProducerX(bus,"Producer",1,10));
     cpu.add(c = new ConsumerX(bus,"Consumer",1,10));
     (void)bus->connect(p,"output",c,"input");
     cpu.fireItUp(5,10000*4196);
}
```

Figure 7: Main entry point for the producer-consumer application

4.1 Writing to the Bus

The bus call used to send a message is

`bus->write(`*thread*`,`*interface_name*`,`*tape*`,`*ret*`,`*from*`,`$w_1, w_2, \ldots, w_n$`)`

where *thread* is a pointer to the sending thread object, *interface_name* is the name of an outgoing interface, *tape* describes the data types of values w_i, and *ret* and *from* are used to transfer port capabilities between threads. Port capabilities are used by threads to reply to messages received on ports with multiple incoming connections. Any thread may write indirectly to the port of another thread given its capability by using NULL for the *interface_name* argument and specifying a non-NULL capability as the *ret* argument. Using capabilities, the software bus can be used to implement procedure call semantics for threads that act as servers to multiple client threads.

4.2 Reading from a named interface

In the simple situation when there is only one interface and its name is known, we can use one bus call each time to read a message on the interface. The call to read a message from a local port is

`bus->read(`*thread*`,`*interface_name*`,`*tape*`,`*atape*`,`*cap*`,`$r_1, \ldots, r_n$`)`

where *thread* is a pointer to the receiving thread object, *interface_name* is the name of the interface, *tape* describes the types of variables r_i, *atape* is a copy of the sender's tape as given in the `bus->write` that originated the message, and *cap* is the capability sent in the `bus->write` call if given. The thread will block at the `bus->read` call until a message arrives on the specified interface or proceeds immediately if a message is already queued on the interface port.

4.3 Reading from any interface

In contrast to the situation where reading from named interfaces is applicable, there are applications in which the messages must be processed as they come in regardless of the interfaces they are received on. In this case, the method

```
iface = (char*)bus->readselect(thread,atape,ret,buffer,sizeof(buffer))
```

is used to receive messages on any incoming local port where *thread* is a pointer to the receiving thread object, *buffer* is declared as a character array, and *iface* is declared as a character pointer. The thread will block at the `bus->readselect` call until a message arrives on any interface, or will proceed immediately if a message is already queued. After the `bus->readselect` call is complete, the *buffer* will contain the message and *iface* will point to the name of the interface where the message arrived. To pull the variables comprising the message from *buffer*, use

$$\texttt{bus->readback(}buffer,tape,ret,r_i,\ldots,r_n\texttt{)}$$

where *tape* describes the message format and each r_i is the address of a variable. The *ret* variable will contain the return capability if provided in the originating `bus->write` invocation.

4.4 Querying the Bus for Messages

A thread can avoid making a blocking read (`bus_read` or `bus_readselect`) by first querying the bus to find out if any messages are queued. To find out how many messages are queued on a particular interface, the method

```
msgs = bus->query_ifmsgs(thread,interface_name)
```

is used where *thread* is a pointer to a thread object and *interface_name* is the name of an interface. The `bus->query_ifmsgs` call returns the number of messages queued. It does not read any messages from the interface, so it is generally followed by a `bus->read` when one or more messages are available. The method

```
nmsgs = bus_query_objmsgs(thread)
```

is used to find out how many messages are queued on all of the thread's interfaces. This method returns the total number of messages queued on all interfaces. It does not read any of these messages, so it is generally followed by a `bus_readselect` when one or more messages are available.

A thread can also be notified if a message arrives on one of its ports by associating the port with a method using the call

```
handle = bus->register(thread,interface_name,method)
```

where *thread* is a pointer to a thread object and *interface_name* is the string name of the local port. When the message arrives on the port, the associated method is invoked. The returned handle allows the program to disassociate the method from the local port during execution using the `bus->deregister` method. Using either `read`, `readselect` or `register`, individual threads can either use polling or interrupts to process incoming messages.

4.5 Implementation

The MINIBUS itself is implemented as a class from which bus instances are created. For each incoming port, the bus allocates a semaphore-protected queue from which `bus->read` and query messages can access. Each queue is an instance of the AWESIME class LOWERBOUNDEDFIFO that can contain a finite number of messages and allows mutually exclusive access to single threads at a time. Since the bus code is reentrant, any call to `bus->write` will enqueue a message on all recipient queues as determined by the bus topology. If the queue is full, the sending thread will block until a message is removed from the queue by the receiving thread. On the other hand, if a queue is empty and the receiving thread attempts to read from the queue, it will block until a message is enqueued. The Awesime class LOWERBOUNDEDFIFO takes care of all synchronization on message queues.

The bus thread has three tasks: enqueue messages, notify recipients, and maintain the topology. The bus thread notifies other threads only if there is a message on a port with a registered method. If a thread has registered a method associated with an incoming port, the bus interrupts the thread by calling this method with the message. Otherwise, the bus thread only enqueues messages and maintains the connection topology.

5 An Example

We present an example application in which the MINIBUS is used for communication between threads: the Dining Philosophers problem. The problem involves the use of parallel threads that rely on message passing. The Dining Philosophers problem is a classic synchronization and resource allocation problem and similar solutions can readily be constructed from this one for other synchronization problems. We present the source code for our solution and discuss our approach to the dining philosophers problem using the MINIBUS.

5.1 The Dining Philosopher's Problem

Our solution to the dining philosophers problem involves *N* philosophers around a table of *N* chopsticks and a shared bowl of rice. Philosophers need two chopsticks to eat, thus only $\lfloor N/2 \rfloor$ philosophers can eat at one time. The problem is to orchestrate the actions of the philosophers to avoid deadlock. For example, if all philosophers grabbed their left chopstick first and refused to release it until they acquired the right chopstick, then the entire table of philosophers would deadlock. This leads to starvation. If each philosopher picks up one chopstick and puts it down if it finds the other chopstick busy, then the philosophers might livelock. Again, this leads to starvation.

In our solution, each philosopher determines which two chopsticks to eat with based on its assigned process identifier and continues a cycle of eating and thinking for a fixed number of mouthfuls. Each philosopher first tries to acquire two chopsticks, eats a mouthful, returns the chopsticks to the table, and thinks for a moment while chewing before repeating the cycle. We define an application class `Phil` as a subclass of `Thread` and a `Table` class to manage the chopsticks. Figure 8 shows the class definition and implementation of class `Phil`. The `Phil` class constructor passes initialization arguments to the `Thread` class in which each `Phil` instance is given the same name. The constructor then changes the thread name to an explicit name based on an identifier number passed to the thread, using the `Thread::name` method. Other variables are also initialized including the instance variable `mouthfulls` that is used to keep track of total consumption of rice by the philosopher. The `Phil::main` member function is the starting point for any thread instance of class `Phil` and it specifies the actions of each philosopher. Each philosopher thread will follow an identical cycle: (a) get two chopsticks; (b) eat a mouthful; (3) return the chopsticks to the table; and (4) think. Step (2) specifies that two chopsticks are acquired but contention is resolved in the `Table` thread.

The virtual functions `getChopsticks` and `returnChopsticks` in the application class Phil are implemented in the interfacing class `PhilX` shown in Figure 9. The member function `getChopsticks` uses a `bus->write` call to request the use of two chopsticks represented by two integers `x` and `y`, on the interface port name "getChopsticks". A capability is created and passed along via the capability argument so that the receiver can respond to the sender indirectly. Since an explicit confirmation of the request is necessary before the philosopher can proceed to eat, a `bus->read` call blocks the philosopher until a reply is received (the reply has no associated data). The pair of commands `bus->write/bus->read` implements procedure call semantics by blocking until a return message granting the request arrives. In contrast, the member function `returnChopsticks` conducts asynchronous communication because it only uses a `bus->write` call to release the two chopsticks.

The `Table` thread acts as a resource manager in the application. Its resources are *N* chopsticks and it knows which chopsticks are available at any given time in order to grant permission to or hold requests to use chopsticks. Figure 10 shows the specification and implementation of the `Table` application class. The main member function continuously calls member functions `getRequest` and `getRelease` to process incoming messages. A `Table` thread will follow the cycle: (1) listen to the bus for request or release messages; (2) take appropriate actions according to the message that is received; and (3) send back a message granting use of chopsticks if it is a request and the request can be met right away. In step (3), the call to the virtual function `grantChopsticks` takes the result of the `getRequest` call as an argument. This is an opaque variable type that contains return information needed in the interfacing layer.

Figure 11 shows the specification and implementation of the interfacing class `TableX`. In this case, the virtual functions `getRequest` and `getRelease` are not implemented with simple `bus->read` operations which would block execution of the `Table` thread. Instead, the method `bus->query_ifmsgs` is used to query for messages on an interface before reading. This approach implements a polling scheme. If a message is not present, a false condition is returned to the caller else the incoming capability is caste into an opaque type (e.g., a pointer to an integer) and returned to the caller in the application layer.

Figure 12 illustrates the application's main function that sets up the systems and connections between the philosopher threads instances and the `Table` thread instance. First, a vector of pointers to philosophers threads is declared. Next, a MINIBUS and CPU are created and the bus is added to the CPU instance. Instances of the philosophers and the table threads are then created and added to the CPU. At this point, connections between thread interface ports are added to the bus topology. Finally, the `fireItUp` member function is invoked on the CPU to start the things going. This solution to the dining philosophers problem can be reconfigured easily with additional philosophers without changes to the specification of any application thread class or its implementation.

5 Discussion

We have used the MINIBUS environment to build simulations of several discrete event systems including the leader election problem, an automobile cruise-control system, a simple avionics guidance, control and navigation platform, a thermostatic control system, a satellite simulation, and a traffic light system. In each case, we are able adjust the numbers of threads between runs of the simulations and change the topology dynamically if necessary.

```
class Phil : public Thread {
     int pid;
     char nameBuffer[128];
     int mouthfulls;
     int nphils;
public:
     Phil(int,int);
     virtual void getChopsticks(int,int);
     virtual void returnChopsticks(int,int);
     void main();
}

Phil::Phil(int xpid,int max,int n) : ("Philosopher",50000,HCCM)
{
     pid = xpid;
     sprintf(nameBuffer,"Phil-%d",xpid);
     name(nameBuffer);
     mouthfulls = max;
     nphils = n;
}

void Phil::main()
{
     int i,next=pid % nphils + 1;
     CERR_ALWAYS_PRE;
     cerr << name() << " using chopsticks " << pid << "," << next << endl;
     CERR_POST;
     for(i=0;i < mouthfulls;i++) {
          getChopsticks(pid,next);
          CERR_ALWAYS_PRE;
          cerr << name() << " eats" << endl;
          CERR_POST;
          returnChopsticks(pid,next);
     }
}
```

Figure 8: Application class specification and implementation for a philosopher thread

```
class PhilX : public Phil {
     Bus bus;
public:
     PhilX(Bus b,int pid,int max,int n):(pid,max,n) {bus = b; bus->init(this);}
     void getChopsticks(int,int);
     void returnChopsticks(int,int);
}

void PhilX::getChopsticks(int a,int b)
{
     Capability cap;
     bus->capability(this,"getChopsticks",&cap);
     bus->write(this,"getChopsticks","II",NULL,&cap,a,b);
     bus->read(this,"getChopsticks",NULL,NULL,NULL);
}
```

Figure 9: Interfacing layer class specification and implementation for philosopher threads

```
class Table : public Thread {
     int nchops;
     Array* chopsticks;
public:
     Table(int);
     virtual int getRequest(int*,int*);
     virtual void grantChopsticks(int,int,int);
     virtual int getRelease(int*,int*);
     virtual int terminate();
}

Table::Table(int size) : ("Table",50000,HCCM)
{
     chopsticks = new Array(size);
     nchops = size;
     for(int i=0;i < size;i++) {
          (*chopsticks)[i] = AVAILABLE;
     }
}
void Table::main()
{
     int x,y,r;
     int count = nchops;
     while((count -= terminate()) {
          if((*chopsticks)[(int)queue1.front()]==AVAILABLE &&
             (*chopsticks)[(int)queue2.front()]==AVAILABLE) {
               x = (int)queue1.front(); y = (int)queue2.front();
               grantChopsticks((int)queueret.front(),x,y);
               queueret.deq(); queue1.deq(); queue2.deq();
               (*chopsticks)[x] = UNAVAILABLE;
               (*chopsticks)[y] = UNAVAILABLE;
          }
          if(r = getRequest(&x,&y)) {
               if((*chopsticks)[x]==AVAILABLE && (*chopsticks)[y]==AVAILABLE) {
                    grantChopsticks(r,x,y);
                    (*chopsticks)[x] = UNAVAILABLE;
                    (*chopsticks)[y] = UNAVAILABLE;
               } else {
                    queueret.enq(r); queue1.enq(x); queue2.enq(y);
               }
          }
          if(getRelease(&x,&y)) {
               (*chopsticks)[x] = AVAILABLE; (*chopsticks)[y] = AVAILABLE;
          }

     }
}
```

Figure 10: Application class specification and implementation for a table thread

```
class TableX : public Table {
     Bus bus;
public:
     TableX(Bus b,int size) : (size) { bus = b; }
}
int TableX::getRequest(int* x,int* y)
{    Capability* cap;
     if(bus->query_ifmsgs(this,"getRequest")) {
          bus->read(this,"getRequest","ii",NULL,cap,x,y); return cap;
     } else return 0;
}
TableX::grantChopsticks(int cap,int x,int y)
{
     bus->write(this,NULL,"II",cap,NULL,x,y);
}
TableX::getRelease(int* x,int* y)
{
     if(bus->query_ifmsgs(this,"getRelease")) {
          bus->read(this,"getRelease","ii",NULL,NULL,x,y); return 1;
     } else return 0;
}
```

Figure 11: Interfacing class specification and implementation for a table thread

```
main()
{
     Thread* phil[NPHILS],table;
     MultiFifoMux cpu;

     Bus* bus = new Bus();
     cpu.add(bus);

     cpu.add(table = new TableX(bus,NPHILS));
     for(int i=0;i < NPHILS;i++) {
       cpu.add(phil[i] = new PhilX(bus,i,10,NPHILS));
       (void)bus->connect(p[i],"getChopsticks",table,"getRequest");
       (void)bus->connect(p[i],"returnChopsticks",table,"getRelease");
       (void)bus->connect(p[i],"terminate",table,"terminate");
     }
     cpu.fireItUp(5,10000*4196);
}
```

Figure 12: Main function for dining philosophers application

For example, consider a sample solution to the leader election problem consisting of four identical threads T1, T2, T3, and T4. All threads are connected together in a ring topology. All threads start by flipping a coin to decide if it runs. If a thread decides to run, it sends its process identifier to the next thread in the ring, else it does nothing. Each thread then waits for messages shifted between threads to arrive from the previous thread in the ring and compares the received value to the maximum value received so far including its own value if it had decided to run. If the received value is greater than the current maximum, the thread saves the value and passes it to the next thread. If the received value is smaller, the thread does nothing. It continues this procedure for N shifts where N is the total number of threads. At the end of this process, each thread should contain the same maximum value and its prints out this result. We can create more complex configurations of the leader election solution by changing the topology, the yielding method, and the number of rounds.

6 Summary

Our research group built the lightweight software bus as a testbed for running experiments on simulations of parallel, discrete-event systems. The MINIBUS approach allows developers of discrete-event simulations to more easily reconfigure and reuse components in a wide variety of configurations. This capability is often needed in experiments that must run many different types of simulations. In conjunction with the AWESIME library, we can adjust the number of CPUs, the scheduling algorithms, the queue sizes, and other parameters in order to evaluate simulation performance. The bus approach also allows developers to monitor and debug parallel systems by "splicing" components along connections that watch the messages between threads.

We are using the splicing technique to introduce monitors that determine whether or not the behavior of the system is correct with respect to formal specifications. By attaching monitors to connections with assertions concerning ordering of message events, we can test whether execution patterns are legal. This method is similar to code annotations [8], but we have extended it to a softwrae bus environment.

Finally, we developed the layered approach in order to distribute threads across actual physical processors without the need for source code changes. It is possible to replace the interfacing classes with the distributed version of the Polylith software bus without making changes to the application specifications and implementations. There still exists some limitations to distribution, but our goal is to provide developers with the ability to reconfigure their systems in distributed environments without major changes to their components. To achieve this goal, we have used a code generator to produce the interfacing layer code automatically from interface specifications written in a modular interconnection language (MIL) [9]. The application's main entry function that creates the Cpu, threads, and connections can also be generated from MIL specifications. We continue to explore this mechanism as a means of increasing the flexibility and reuse of discrete-event simulations.

7 References

[1] Purtilo, J., Polylith: An Environment to Support Management of Tool Interfaces, ACM SIGPLAN Symposium on Language Issues in Programming Environments, Seattle, WA, June 25-28 1985, pp. 12-18.

[2] Grunwald, D., User's Guide to A Widely Extensible Simulation Environment, University of Colorado, Boulder Technical Report.

[3] Notkin, D., Proxies: A Software Structure for Accommodating Heterogeneity, Software Practice and Experience, April 1990, Volume 20, Number 4, pp. 357-364.

[4] Magee, J. and J. Kramer, Constructing Distributed Systems in Conic, IEEE Transactions on Software Engineering, June 1989, Volume 15, Number 6, pp. 663-675.

[5] Hoare, C.A.R., Communicating Sequential Processes, Prentice Hall, Englewood Cliffs, NJ, 1985.

[6] Garlan, D., G. Kaiser, D. Notkin, Using Tool Abstraction to Compose Systems, IEEE Computer, June 1992, Volume 25, Number 6, pp. 30-38.

[7] Hofmeister, C. R., Dynamic Reconfiguration in Software Buses, University of Maryland Computer Science Department, August 1993.

[8] Belz, F. and D. Luckham, A new approach to prototyping Ada-based hardware/software systems. In *Proceedings of the ACM Tri-Ada Conference*, Baltimore, MD, December 1990.

[9] J. Callahan and J. Purtilo, A packaging system for heterogeneous execution environments, *IEEE Transactions on Software Engineering*, June 1991, Volume 17, Number 6, pp. 626-635.